developer.*
The Independent Magazine for Software Professionals

# Code as Design: Three Essays by Jack W. Reeves

## Introduction

The following essays by Jack W. Reeves offer three perspectives on a single theme, namely that programming is fundamentally a design activity and that the only final and true representation of "the design" is the source code itself. This simple assertion gives rise to a rich discussion—one which Reeves explores fully in these three essays.

The first essay, "What Is Software Design?," was first published in the Fall 1992 issue of the now defunct *C++ Journal*. After a period of obscurity, in recent years the essay has entered the flow of ideas and discussion in the software development community at large, largely due to its exposure on the web and in Robert Martin's book *Agile Software Development: Principles, Patterns, and Practices*. The essay is published here in its entirety and in its original form.

The second essay, "What Is Software Design: 13 Years Later," is the author's first writing on the subject of "code as design" since 1992. Over the years Reeves has followed the discussion that grew from the original essay, and here for the first time he responds to the most common arguments he has encountered. In the course of this task, Reeves also considers certain ideas from the original essay in light of more current trends and techniques.

The third essay, titled simply "Letter to the Editor," is the original letter written by Jack W. Reeves to *C++ Journal*. The letter is published here for the first time. It stands as a rewarding essay in its own right, giving first written expression to the themes and ideas found in "What Is Software Design?" In some aspects it is even more comprehensive and spirited than the essay it inspired.

In the thirteen years since their original publication, these ideas—even today still fresh, challenging, and controversial—have taken hold (in different ways and to differing degrees) as key concepts underlying new, increasingly popular techniques and schools of thought such as Agile and craft-based methodologies; test-first design and test-driven development; and refactoring.

# What Is Software Design?

By Jack W. Reeves

Object oriented techniques, and C++ in particular, seem to be taking the software world by storm. Numerous articles and books have appeared describing how to apply the new techniques. In general, the questions of whether O-O techniques are just hype have been replaced by questions of how to get the benefits with the least amount of pain. Object oriented techniques have been around for some time, but this exploding popularity seems a bit unusual. Why the sudden interest? All kinds of explanations have been offered. In truth, there is probably no single reason. Probably, a combination of factors has finally reached critical mass and things are taking off. Nevertheless, it seems that C++ itself is a major factor in this latest phase of the software revolution. Again, there are probably a number of reasons why, but I want to suggest an answer from a slightly different perspective: C++ has become popular because it makes it easier to design software and program at the same time.

If that comment seems a bit unusual, it is deliberate. What I want to do in this article is take a look at the relationship between programming and software design. For almost 10 years I have felt that the software industry collectively misses a subtle point about the difference between developing a software design and what a software design really is. I think there is a profound lesson in the growing popularity of C++ about what we can do to become better software engineers, if only we see it. This lesson is that programming is not about building software; programming is about designing software.

Years ago I was attending a seminar where the question came up of whether software development is an engineering discipline or not. While I do not remember the resulting discussion, I do remember how it catalyzed my own thinking that the software industry has created some false parallels with hardware engineering while missing some perfectly valid parallels. In essence, I concluded that we are not software engineers because we do not realize what a software design really is. I am even more convinced of that today.

The final goal of any engineering activity is the some type of documentation. When a design effort is complete, the design documentation is turned over to the manufacturing team. This is a completely different group with completely different skills from the design team. If the design documents truly represent a complete design, the manufacturing team can proceed to build the product. In fact, they can proceed to build lots of the product, all without any further intervention of the designers. After reviewing the software development life cycle as I understood it, I concluded that the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code listings.

There are probably enough arguments both for and against this premise to fill numerous articles. This article assumes that final source code is the real software design and then examines some of the consequences of that assumption. I may not be able to prove that this point of view is correct, but I hope to shown that it does explain some of the observed facts of the software industry, including the popularity of C++.

There is one consequence of considering code as software design that completely overwhelms all others. It is so important and so obvious that it is a total blind spot for most software organizations. This is the fact that software is cheap to build. It does not qualify as inexpensive; it is so cheap it is almost free. If source code is a software design, then actually building software is done by compilers and linkers. We often refer to the process of compiling and linking a complete software system as "doing a build". The capital investment in software construction equipment is low—all it really takes is a computer, an editor, a compiler, and a linker. Once a build environment is available, then actually doing a software build just takes a little time. Compiling a 50,000 line C++ program may seem to take forever, but how long would it take to build a hardware system that had a design of the same complexity as 50,000 lines of C++.

Another consequence of considering source code as software design is the fact that a software design is relatively easy to create, at least in the mechanical sense. Writing (i.e., designing) a typical software module of 50 to 100 lines of code is usually only a couple of day's effort (getting it fully debugged is another story, but more on that later). It is tempting to ask if there is any other engineering discipline that can produce designs of such complexity as software in such a short time, but first we have to figure out how to measure and compare complexity. Nevertheless, it is obvious that software designs get very large rather quickly.

Given that software designs are relatively easy to turn out, and essentially free to build, an unsurprising revelation is that software designs tend to be incredibly large and complex. This may seem obvious but the magnitude of the problem is often ignored. School projects often end up being several thousand lines of code. There are software products with 10,000 line designs that are given away by their designers. We have long since passed the point where simple software is of much interest. Typical commercial software products have designs that consist of hundreds of thousands of lines. Many software designs run into the millions. Additionally, software designs are almost always constantly evolving. While the current design may only be a few thousand lines of code, many times that may actually have been written over the life of the product.

While there are certainly examples of hardware designs that are arguably as complex as software designs, note two facts about modern hardware. One, complex hardware engineering efforts are not always as free of bugs as software critics would have us believe. Major microprocessors have been shipped with errors in their logic, bridges collapsed, dams broken, airliners fallen out of the sky, and thousands of automobiles and other consumer products have been recalled - all within recent memory and all the result of design errors. Second, complex hardware designs have correspondingly complex and expensive build phases. As a result, the ability to manufacture such systems limits the number of companies that produce truly complex hardware designs. No such limitations exist for software. There are hundreds of software organizations, and thousands of very complex software systems in existence. Both the number and the complexity are growing daily. This means that the software industry is not likely to find solutions to its problems by trying to emulate hardware developers. If anything, as CAD and CAM systems have helped hardware designers to create more and more complex designs, hardware engineering is becoming more and more like software development.

Designing software is an exercise in managing complexity. The complexity exists within the software design itself, within the software organization of the company, and within the industry as a whole. Software design is very similar to systems design. It can span multiple technologies and often involves multiple sub-disciplines. Software specifications tend to be fluid, and change rapidly and often, usually while the design process is still going on. Software development teams also tend to be fluid, likewise often changing in the middle of the design process. In many ways, software bears more resemblance to complex social or organic systems than to hardware. All of this makes software design a difficult and error prone process. None of this is original thinking, but almost 30 years after the software engineering revolution began, software development is still seen as an undisciplined art compared to other engineering professions.

The general consensus is that when real engineers get through with a design, no matter how complex, they are pretty sure it will work. They are also pretty sure it can be built using accepted construction techniques. In order for this to happen, hardware engineers spend a considerable amount of time validating and refining their designs. Consider a bridge design, for example. Before such a design is actually built the engineers do structural analysis; they build computer models and run simulations; they build scale models and test them in wind tunnels or other ways. In short, the designers do everything they could think of to make sure the design is a good design before it is built. The design of new airliner is even worse; for those, full scale prototypes must be built and test flown to validate the design predictions.

It seems obvious to most people that software designs do not go through the same rigorous engineering as hardware designs. However, if we consider source code as design, we see that software designers actually do a considerable amount of validating and refining their designs. Software designers do not call it engineering, however, we call it testing and debugging. Most people do not consider testing and debugging as real "engineering"; certainly not in the software business. The reason has more to do with the refusal of the software industry to accept code as design than with any real engineering difference. Mock-ups, prototypes, and bread-boards are actually an accepted part of other engineering disciplines. Software designers do not have or use more formal methods of validating their designs because of the simple economics of the software build cycle.

Revelation number two: it is cheaper and simpler to just build the design and test it than to do anything else. We do not care how many builds we do—they cost next to nothing in terms of time, and the resources used can be completely reclaimed later if we discard the build. Note that testing is not just concerned with getting the current design correct, it is part of the process of refining the design. Hardware engineers of complex systems often build models (or at least they visually render their designs using computer graphics). This allows them to get a "feel" for the design that is not possible by just reviewing the design itself. Building such a model is both impossible and unnecessary with a software design. We just build the product itself. Even if formal software proofs were as automatic as a compiler, we would still do build/test cycles. Ergo, formal proofs have never been of much practical interest to the software industry.

This is the reality of the software development process today. Ever more complex software designs are being created by an ever increasing number of people and organizations. These designs will be coded in some programming language and then validated and refined via the build/test cycle. The process is error prone and not particularly rigorous to begin with. The fact that a great many software developers do not want to believe that this is the way it works compounds the problem enormously.

Most current software development processes try to segregate the different phases of software design into separate pigeon-holes. The top level design must be completed and frozen before any code is written. Testing and debugging are necessary just to weed out the construction mistakes. In between are the programmers, the construction workers of the software industry. Many believe that if we could just get programmers to quit "hacking" and "build" the designs as given to them (and in the process, make fewer errors) then software development might mature into a true engineering discipline. Not likely to happen as long as the process ignores the engineering and economic realities.

For example, no other modern industry would tolerate a rework rate of over 100% in its manufacturing process. A construction worker who can not build it right the first time, most of the time, is soon out of a job. In software, even the smallest piece of code is likely to be revised or completely rewritten during testing and debugging. We accept this sort of refinement during a creative process like design, not as part of a manufacturing process. No one expects an engineer to create a perfect design the first time. Even if she does, it must still be put through the refinement process just to prove that it was perfect.

If we learn nothing else from Japanese management techniques, we should learn that it is counter-productive to blame the workers for errors in the process. Instead of continuing to force software development to conform to an incorrect process model, we need to revise the process so that it helps rather than hinders efforts to produce better software. This is the litmus test of "software engineering." Engineering is about how you do the process, not about whether the final design document needs a CAD system to produce it.

The overwhelming problem with software development is that *everything* is part of the design process. Coding is design, testing and debugging are part of design, and what we typically call software design is still part of design. Software may be cheap to build, but it is incredibly expensive to design. Software is so complex that there are plenty of different design aspects and their resulting design views. The problem is that all the different aspects interrelate (just like they do in hardware engineering). It would be nice if top level designers could ignore the details of module algorithm design. Likewise, it would be nice if programmers did not have to worry about top level design issues when designing the internal algorithms of a module. Unfortunately, the aspects of one design layer intrude into the others. The choice of algorithms for a given module can be as important to the overall success of the software system as any of the higher level design aspects. There is no hierarchy of importance among the different aspects of a software design. An incorrect design at the lowest module level can be as fatal as a mistake at the highest level. A software design must be complete and correct in all its aspects, or all software builds based on the design will be erroneous.

In order to deal with the complexity, software is designed in layers. When a programmer is worrying about the detailed design of one module, there are probably hundreds of other modules and thousands of other details that he can not possibly worry about at the same time. For example, there are important aspects of software design that do not fall cleanly into the categories of data structures and algorithms. Ideally, programmers should not have to worry about these other aspects of a design when designing code.

This is not how it works, however, and the reasons start to make sense. The software design is not complete until it has been coded *and* tested. Testing is a fundamental part of the design validation and refinement process. The high level structural design is not a complete software design; it is just a structural framework for the detailed design. We have very limited capabilities for rigorously validating a high level design. The detailed design will ultimately influence (or *should* be allowed to influence) the high level design at least as much as other factors. Refining all the aspects of a design is a process that should be happening throughout the design cycle. If any aspect of the design is frozen out of the refinement process, it is hardly surprising that the final design will be poor or even unworkable.

It would be nice if high level software design could be a more rigorous engineering process, but the real world of software systems is not rigorous. Software is too complex and it depends on too many other things. Maybe some hardware does not work quite the way the designers thought it did, or a library routine has an undocumented restriction. These are the kinds of problems that every software project encounters sooner or later. These are the kinds of problems discovered during testing (if we do a good job of testing), for the simple reason that there was no way to discover them earlier. When they are discovered, they force a change in the design. If we are lucky, the design changes are local. More often than not, the changes will ripple through some significant portion of the entire software design (Murphy's Law). When part of the effected design can not change for some reason, then the other parts of the design will have to be weakened to accommodate. This often results is what managers perceive as "hacking", but it is the reality of software development.

For example, I recently worked on a project where a timing dependency was discovered between the internals of module A and another module B. Unfortunately, the internals of module A were hidden behind an abstraction that did not permit any way to incorporate the invocation of module B in its proper sequence. Naturally, by the time the problem was discovered, it was much too late to try to change the abstraction of A. As expected, what happened was an increasingly complex set of "fixes" applied to the internal design of A. Before we finished installing version 1, there was the general feeling that the design was breaking down. Every new fix was likely to break some older fix. This is a normal software development project. Eventually, my colleagues and I argued for a change in the design, but we had to volunteer free overtime in order to get management to agree.

On any software project of typical size, problems like these are guaranteed to come up. Despite all attempts to prevent it, important details will be overlooked. This is the difference between craft and engineering. Experience can lead us in the right direction. This is craft. Experience will only take us so far into uncharted territory. Then we must take what we started with and make it better through a controlled process of refinement. This is engineering.

As just a small point, all programmers know that writing the software design documents after the code instead of before, produces much more accurate documents. The reason is now obvious. Only the final design, as reflected in code, is the only one refined during the build/test cycle. The probability of the initial design being unchanged during this cycle is inversely related to the number of modules and number of programmers on a project. It rapidly becomes indistinguishable from zero.

In software engineering, we desperately need good design at all levels. In particular, we need good top level design. The better the early design, the easier detailed design will be. Designers should use anything that helps. Structure charts, Booch diagrams, state tables, PDL, etc.—if it helps, then use it. We must keep in mind, however, that these tools and notations are not a software design. Eventually, we have to create the real software design, and it will be in some programming language. Therefore, we should not be afraid to code our designs as we derive them. We simply must be willing to refine them as necessary.

There is as yet no design notation equally suited for use in both top level design and detailed design. Ultimately, the design will end up coded in some programming language. This means that top level design notations have to be translated into the target programming language before detailed design can begin. This translation step takes time and introduces errors. Rather than translate from a notation that may not map cleanly into the programming language of choice, programmers often go back to the requirements and redo the top level design, coding it as they go. This, too, is part of the reality of software development.

It is probably better to let the original designers write the original code, rather than have someone else translate a language independent design later. What we need is a unified design notation suitable for all levels of design. In other words, we need a programming language that is also suitable for capturing high level design concepts. This is where C++ comes in. C++ is a programming language suitable for real world projects that is also a more expressive software design language. C++ allows us to directly express high level information about design components. This makes it easier to produce the design, and easier to refine it later. With its stronger type checking, it also helps the process of detecting design errors. This results in a more robust design, in essence a better engineered design.

Ultimately, a software design must be represented in some programming language, and then validated and refined via a build/test cycle. Any pretense otherwise is just silliness. Consider what software development tools and techniques have gained popularity. Structured programming was considered a breakthrough in its time. Pascal popularized it and in turn became popular. Object oriented design is the new rage and C++ is at the heart of it. Now think about what has not worked. CASE tools? Popular, yes; universal, no.

Structure charts? Same thing. Likewise, Warner-Orr diagrams, Booch diagrams, object diagrams, you name it. Each has its strengths, and a single fundamental weakness—it really isn't a software design. In fact the only software design notation that can be called widespread is PDL, and what does that look like.

This says that the collective subconscious of the software industry instinctively knows that improvements in programming techniques and real world programming languages in particular are overwhelmingly more important than anything else in the software business. It also says that programmers are interested in design. When more expressive programming languages become available, software developers will adopt them.

Also consider how the process of software development is changing. Once upon a time we had the waterfall process. Now we talk of spiral development and rapid prototyping. While such techniques are often justified with terms like "risk abatement" and "shortened product delivery times", they are really just excuses to start coding earlier in the life cycle. This is good. This allows the build/test cycle to start validating and refining the design earlier. It also means that it is more likely that the software designers that developed the top level design are still around to do the detailed design.

As noted above—engineering is more about how you do the process than it is about what the final product looks like. We in the software business are close to being engineers, but we need a couple of perceptual changes. Programming and the build/test cycle are central to the process of engineering software. We need to manage them as such. The economics of the build/test cycle, plus the fact that a software system can represent practically anything, makes it very unlikely that we will find any general purpose methods for validating a software design. We can improve this process, but we can not escape it.

One final point: the goal of any engineering design project is the production of some documentation. Obviously, the actual design documents are the most important, but they are not the only ones that must be produced. Someone is eventually expected to use the software. It is also likely that the system will have to be modified and enhanced at a later time. This means that auxiliary documentation is as important for a software project as it is for a hardware project. Ignoring for now users manuals, installation guides, and other documents not directly associated with the design process, there are still two important needs that must be solved with auxiliary design documents.

The first use of auxiliary documentation is to capture important information from the problem space that did not make it directly into the design. Software design involves inventing software concepts to model concepts in a problem space. This process requires developing an understanding of the problem space concepts. Usually this understanding will include information that does not directly end up being modeled in the software space,

but which nevertheless helped the designer determine what the essential concepts were, and how best to model them. This information should be captured somewhere in case the model needs to be changed at a later time.

The second important need for auxiliary documentation is to document those aspects of the design that are difficult to extract directly from the design itself. These can include both high level and low level aspects. Many of these aspects are best depicted graphically. This makes them hard to include as comments in the source code. This is *not* an argument for a graphical software design notation instead of a programming language. This is no different from the need for textual descriptions to accompany the graphical design documents of hardware disciplines. Never forget that the source code determines what the actual design really is, not the auxiliary documentation. Ideally, software tools would be available that post processed a source code design and generated the auxiliary documentation. That may be too much to expect. The next best thing might be some tools that let programmers (or technical writers) extract specific information from the source code that can then be documented in some other way. Undoubtedly, keeping such documentation up to date manually is difficult. This is another argument for the need for more expressive programming languages. It is also an argument for keeping such auxiliary documentation to a minimum and keeping it as informal as possible until as late in the project as possible. Again, we could use some better tools, otherwise we end up falling back on pencil, paper, and chalk boards.

To summarize:

- Real software runs on computers. It is a sequence of ones and zeros that is stored on some magnetic media. It is not a program listing in C++ (or any other programming language).
- A program listing is a document that represents a software design. Compilers and linkers actually build software designs.
- Real software is incredibly cheap to build, and getting cheaper all the time as computers get faster.
- Real software is incredibly expensive to design. This is true because software is incredibly complex and because practically all the steps of a software project are part of the design process.
- Programming is a design activity—a good software design process recognizes this and does not hesitate to code when coding makes sense.
- Coding actually makes sense more often than believed. Often the process of rendering the design in code will reveal oversights and the need for additional design effort. The earlier this occurs, the better the design will be.

- Since software is so cheap to build, formal engineering validation methods are not of much use in real world software development. It is easier and cheaper to just build the design and test it than to try to prove it.
- Testing and debugging are design activities—they are the software equivalent of the design validation and refinement processes of other engineering disciplines. A good software design process recognizes this and does not try to short change the steps.
- There are other design activities—call them top level design, module design, structural design, architectural design, or whatever. A good software design process recognizes this and deliberately includes the steps.
- All design activities interact. A good software design process recognizes this and allows the design to change, sometimes radically, as various design steps reveal the need.
- Many different software design notations are potentially useful—as auxiliary documentation and as tools to help facilitate the design process. They are not a software design.
- Software development is still more a craft than an engineering discipline. This is primarily because of a lack of rigor in the critical processes of validating and improving a design.
- Ultimately, real advances in software development depend upon advances in programming techniques, which in turn mean advances in programming languages. C++ is such an advance. It has exploded in popularity because it is a mainstream programming language that directly supports better software design.
- C++ is a step in the right direction, but still more advances are needed.

###



*<continued next page>*

# What Is Software Design: 13 Years Later

By Jack W. Reeves

People have occasionally asked whether I did any follow-on writing to my “What Is Software Design” article. The answer has basically been “No, not really.” I want to make it clear that that is not because I forgot about it or otherwise changed my mind. Allow me to offer a bit of explanation.

When the article appeared, I hoped–actually expected–that I would get some type of rebuttal from some sort of industry “expert.” I was looking forward to this since part of my reason for writing the article had been hopes of stimulating discussion within the software industry about the overall software development process. Nothing happened.

There were no letters to the editor that I know about and nothing ever sent directly to me. *C++ Journal* became defunct shortly after that issue, and I figured my article had gone to that great land fill in the sky that swallows most publications. I went on to doing other things. It wasn't until 1997 or 1998 that I got an email from Bob Martin (who had just taken over as editor of the *C++ Report*) letting me know there was a wiki page about my article on Ward Cunningham's `c2.com` web site. This was–quite literally–the first time I knew anybody had read my article (other than the people I personally gave a copy to).

I started to follow the discussions on the wiki page and occasionally on some news groups, but deliberately stayed out of them myself for several reasons: a) I was focused on certain other things at the time, b) it was pretty obvious that other people who had accepted what I was trying to say were just as qualified–maybe more so–to argue the points as I would have been (I specifically remember Michael Feathers writing), and c) last but not least, it still looked to me like there was a lot of opposition to the concept. Unfortunately, most of the arguments sounded pretty much like the ones I had been dealing with for almost 15 years by that point (remember, I had had the idea almost 10 years before I wrote the article).

I had grown tired of trying to deal with people who were totally incapable of getting past their own pre-conceptions to even consider the idea rationally. It was like trying to explain that the French speak a different language to someone who is convinced that “different language” really just means “different dialect of English”. No matter what you say, they will parse your arguments against their beliefs and either dismiss you out of hand, or patronize you with their counter arguments. I had seen a number of projects where “design it in the code” worked, but even the people on such projects often refused to accept the reality. My level of cynicism about being able to improve things was very high.

It still is, but I think it is time I made some attempt to actually defend myself, rather than let other people do it. Therefore, what I am going to do is address some of the most common criticisms I have seen about “What Is Software Design?”.

A. Initially, the most common criticism I would see can be summarized as “If source code is the design, then programmers are designers; but obviously they are not, therefore source code cannot be the design.” Nobody states it that baldly, but when you parse what they do say, it comes down to the same thing. These are circular arguments that start with the assumption that programming/coding is a manufacturing type of activity. In logic, this is known as a “Begging the Question” fallacy. In essence, these people say “your assumption (i.e. source code is the design) contradicts my assumption (i.e. programmers are assembly workers), therefore your assumption must be wrong.”

Someone might suggest that I am doing the same thing, i.e. starting with the assumption that source code is a design. I accept that–up to a point. While I will admit that a lot of the article reads like an attempt to prove that “source code is the design”, that was not really what I was trying to do. The following quote is from the beginning of the article:

> “This article assumes that final source code is the real software design and then examines some of the consequences of that assumption. I may not be able to prove that this point of view is correct, but I hope to show that it does explain some of the observed facts of the software industry, ...”

I did not set out to prove that “source code is the design”; I will readily concede that what is a “design” is to some extent a matter of definition. The point of the article was to try to show how this assumption led to much better explanations of numerous observed *facts*. I am still waiting for anyone to offer better explanations based upon alternative assumptions.

B. These days, thanks in part to the rise of Extreme Programming and other Agile Methods, people are starting to accept (grudgingly) that programmers are not assembly line drones. Unfortunately, that doesn’t mean they are willing to accept the concept of “the source code is the design”. The arguments can be summarized by the example that is still on the wiki page:

> “As for throwing the whole design thing out, and just designing in code... Hahahahahahahahah no really Hahahahahahahahah :)”

This really makes me angry. For reasons that I do not understand, reasonably intelligent people insist upon confusing the concept of design as *process* versus design as *product*. You would think that anyone who passed high school would understand the difference between the process of writing a paper (for example) and the paper itself. Certainly, you would expect anyone with a college background to understand that there are often lots of different ways to arrive at the same solution.

Nevertheless, people keep insisting that my contention of "the source code is the design" means "don't do design, just code." I never said anything of the sort. What I did say was:

> "In software engineering, we desperately need good design at all levels. In particular, we need good top level design. The better the early design, the easier detailed design will be. Designers should use anything that helps. Structure charts, Booch diagrams, state tables, PDL, etc.—if it helps, then use it."

Today, I would phrase it differently. I would say we need good architectures (top level design), good abstractions (class design), and good implementations (low level design). I would also say something about using UML diagrams or CRC cards to explore alternatives. Nevertheless, I will not back away from the following statement:

> "We must keep in mind, however, that these tools and notations are not a software design. Eventually, we have to create the real software design, and it will be in some programming language. Therefore, we should not be afraid to code our designs as we derive them."

This is fundamental. I am not arguing that we should not "do design." However you want to approach the *process*, I simply insist that you have not completed the process until you have written *and* tested the code.

Personally, I think a person with his feet on the desk staring at the ceiling can be "doing design" just as seriously as someone playing with UML diagrams in ROSE. I have always known that you are better off if you put some real thought into what you are trying to do before actually doing it. People differ widely in what helps them think, however. Some people use pencil and paper. Others like white boards or even computer tools. Some people like to bounce ideas off of other people, others like peace and quiet. Some people feel comfortable with diagrams like UML. Others prefer CRC cards.

What approach they choose doesn't matter; until someone starts insisting that these intermediate designs should be products in their own right. It's the code that matters. If you get good code, does it really matter how it came about? If you don't get good code, does it really matter how much other garbage you made people do before they wrote the bad code?

Everybody that has been in this business any length of time has seen plenty of examples where someone obviously sat down and coded the first thing that popped into their mind. Later, when it became obvious that there were shortcomings with the approach, there was too much blood, sweat, and "skin" invested in the code to scrap it and do something better. Fine, we all know a little thought can go a long way.

On the other hand, any of us who has spent time on a *traditional* development project with its strict rules forbidding the writing of a single line of code until the "design" is completed and reviewed and approved, etc. knows you can waste a hell of a lot of time producing documents that are out of date literally days after the actual coding starts. Why bother?

You think we could find some happy medium of "enough" design effort, but not too much. There is no such thing. The only way we validate a software design is by building it and testing it. There is no silver bullet, and no "right way" to do design. Sometimes an hour, a day, or even a week spent thinking about a problem can make a big difference when the coding actually starts. Other times, 5 minutes of testing will reveal something you never would have thought about no matter how long you tried. We do the best we can under the circumstances, and then refine it.

One last comment: I also did not say that the only necessary documentation is the source code. I specifically pointed out in the article:

> "...auxiliary documentation is as important for a software project as it is for a hardware project."

The source code may be the master design document, but it is seldom the only one necessary.

B′. I cannot resist making a remark about a side issue that often comes up in discussions about Extreme Programming and Agile methods that is related to the above. This is often phrased as a question: what about the Less Able Programmer? The issue seems to be that only the very best programmers can "design" and "code" at the same time. To offset this, we must have all those intermediate design steps and products mentioned above to make up for the lack of experience and talent of the average programmer.

To me, this is like asking “what do we do about the less able physician?” I know the practice of medicine and software development are not analogous, but bear with me for a moment. An awful lot of the practice of medicine is pretty much rote (we joke about “take two aspirin and call me tomorrow”). Nevertheless, the medical profession still insists upon some pretty high standards of intelligence, education, and experience before someone is allowed to call themselves an MD. In other words, we want our doctors to know what they are doing.

In software development, questions about the less able programmer really come down to trying to substitute a process for intelligence, aptitude, and experience. Apparently, a lot of people think that if we force people to create enough UML diagrams (or whatever), have enough reviews, and otherwise follow a detailed process, that eventually they will figure out what they are doing and code it correctly. There is no evidence that such approaches have worked in the past, and I see no reason to believe they will work in the future. In fact, my own experience says that properly using tools such as UML involves a considerable level of expertise and experience in its own right.

C. Another argument I have seen questions the contention that the goal of an engineering effort is some type of documentation. Some people argue that the goal of engineering is a “product” and that real engineers often “build” things and those “things” are as much a product of engineering as any documentation.

This argument tries to sidestep the question of “What is a Software Design?” by implying a parallel between the “things” that other engineers build and what software developers create. Frankly, this is nonsense. I will concede that there are engineers who build “things” with little or no formal design documentation, and I suspect that even in those cases there is probably some design documentation (even if it is on the back of an envelope). In any case, I think we can safely say that such projects produce only one-off products and are usually done by individuals.

When an “engineering” effort starts involving more than a couple of people, or when it has a formal manufacturing phase, then documentation starts to loom larger and larger as the actual product of the engineering effort. You better believe that the engineers at Toyota or Motorola produce documentation, and we’ll not even think about the engineers at Boeing or Lockheed. So, while it might be true that a lot of engineers do things besides producing design documents, anyone who calls himself an engineer knows what a design document in his field looks like, and probably produces such more often than not. Can we say the same for “software engineers”?

Incidentally, this contention regarding engineers and documentation was not mine originally, but instead something I picked up from an article in *Datamation* back in 1979. I agree with it completely however.

D. One final but fairly minor argument I have seen is that source code is still too high level to be a design. At least one critic wanted to call source code the "specification." His (or her) take was that the real design is what comes out of the compiler. In some sense this is just a matter of definition, but I still disagree with it.

The generally accepted definition is that a "specification" states the *what*, which is followed by a design document that details the *how*. While there is a certain amount of flexibility allowed of the compiler in determining the *how* of object code, there is certainly no creativity involved. And that is where I draw the line. When the document is detailed enough, complete enough, and unambiguous enough that it can be interpreted mechanistically, whether by a computer or by an assembly line worker, then you have a design document. If it still requires creative human interpretation, then you don't.

In software development, the design document is a source code listing.

###



<*continued next page*>

# Letter to the Editor

May 19, 1992

From: Jack W. Reeves, San Jose, CA

To: Livleen Singh, Editor, *C++ Journal*, Port Washington, NY

Dear Editor,

Thank you for printing my letter of August 27, 1991 commenting on software design. I agree (in principal, if not in detail) with most of your reply. What we all want is to find ways to produce better software and help our industry "mature into a disciplined science." My problem is, about ten years ago I came to the conclusion that, as an industry, we do not understand what a software design really is. I am even more convinced of this today. I do not claim that my point of view is correct, but I have found it very useful in explaining why some things work and why others do not. There is a very subtle, but very important point which is being missed. This is the difference between doing software design and what a finished software design really is. I would like to elaborate upon this point.

Allow me to begin with the final part of your reply. I made the statement (supplying context) "It may not be a very good (software) design, but a (software) design it is." You suggest comparing software design with bridge design and created the following statement "It may not be a very good bridge, but a bridge it is." Here the word "bridge" is substituted for "(software) design." The interpretation seems to be "Would you trust something that was built with little or no design?" The obvious answer is "Of course not!" The comparison is not valid, however. The fact that it seems valid to most of the industry is exactly the point I am trying to make.

Instead of changing the sentence, change the context instead. Now the statement would read "It may not be a very good (bridge) design, but a (bridge) design it is." Would you volunteer to be the first across this bridge? The immediate answer might be "No, a bad design is no better than no design!" A little thought will show that an equally valid answer is "What bridge?" Until you actually build the bridge from the design there is no need to worry about crossing it. The point is that we don't build bridges from scratch designs. Before the bridge actually gets built, the design will be refined considerably. We will do analysis. We will build computer models and run simulations. We may even build scale models and test them in wind tunnels and other ways. We will do everything we can to make sure the design is a good design before we build the bridge. We call this "engineering" (and sometimes, despite everything, it still isn't completely right...there was this bridge in Tacoma).

Back to software. We in the software industry also refine our designs, only we don't get to call it engineering. We call it "testing and debugging". This phase of the software lifecycle takes a long time. All too often it takes longer than planned. Unfortunately, it is often not enough and the final designs that we turn into deliverable software are still not as good as they should be. This seems like a fact of software life. Many people lament it and ask why we software developers do not "engineer" our designs better? Many explanations are offered, but never the one most obvious to me -- simple economics. Software is dirt cheap to build.

Am I crazy? I don't think so! Compiling and linking 50,000 lines of C++ code on your 486 may seem to take forever, but how would you like to assemble a circuit card with 50,000 discreet components, or build a bridge with 50,000 structural elements? We don't construct mathematical proofs of software correctness or run our code through symbolic executors because it takes less time and effort to just build it and test it. We probably would get better software if we did more of the former, but we don't. Why not?

There are probably lots of reasons, but I would like to suggest that many of them derive from our failure to consider testing and debugging as part of the software design process. We would like for it to go away completely. Since it will not, we try to treat it as some sort of "quality assurance" function and spend as little time, effort, and money on it as we can get away with. We consider it a shame of the software industry that testing and debugging take up half the typical software development lifecycle. Is it really so bad? I suspect that most engineers in other disciplines haven't a clue about what percentage of their time is spent actually creating a design and what is spent on testing and debugging the result. Some industries are probably better than software. I am pretty sure that other industries are actually much worse. Consider what it must take to "design" a new airliner.

I get somewhat testy when people start making gratuitous comparisons between software design and other engineering disciplines. Major microprocessors have been shipped with bugs in their logic, bridges have collapsed, dams broken, airliners fallen out of the sky, and thousands of automobiles and other consumer products recalled - all within recent memory and all the result of design errors.

The problem with software is - design is not just important, it is basically everything. Saying that programmers should not have to design is like saying fish should not have to swim. When I am coding, I am designing. I am creating a software design out of the void. Sometimes the algorithm is simple and the design is trivial. Often times, I have to design data structures and the algorithms to match. There may be alternatives and I have to choose between them based upon my perception of the advantages and disadvantages of each. Sometimes I decide that the design is getting too complex and I specify one or more

sub-modules. When I have finished the design, I test it to see how good it is and refine it to make it better. Refinements not only come from finding errors in the original design, but from other sources such as peer walkthroughs and formal reviews. The bottom line is that my design must be correct, or every piece of software built from it will be erroneous. Therefore I concentrate on doing it right, and it takes mental effort and skill, just like any other creative design activity.

Nevertheless, most software systems are quite large and quite complex and my one module is only a small part. While I am concentrating on the details of code design for module X, there may be hundreds of other modules and thousands of other details that I can not possibly worry about at the same time. There are also important aspects of software design which do not fall cleanly into the categories of data structures and algorithms. It is these "other aspects" that most people mean when they say software design.

It is true that programmers do not want to worry about the "high level aspects" of a design when they are designing code. They often end up having to worry about them anyway. The high level design clearly affects the detailed design. The converse is also true. The details of internal design may (or *should*) help decide amongst high level alternatives. Refining all aspects of the design is a process that should be happening throughout the development cycle. If some aspect of the design is "frozen" out of the refinement process, you can potentially end up with a poor or even unworkable final design.

Some of my colleagues have interpreted my harangues on this subject as "Jack says forget design and just start coding". Nothing could be further from the truth (though I see how they get that impression). I am not against traditional software design. We desperately need good design at all levels. It doesn't matter whether we call the early process top level design, structural design, module design, or whatever. What I have been arguing for are two changes in perception. First, that we recognize that the results of the early design steps are not a complete software design any more than the first rough sketches are a complete bridge design. Second, that we capture our design thinking using a notation that is a true skeleton software design. That means using a programming language.

Ultimately, the computer doesn't care how we get to a final software design any more than the steeplejack building a bridge cares how the bridge design was refined and validated. All that matters to either of them is that the design they are working from be sufficient to allow them to correctly build the product. On the other hand, what it takes to create a good software design obviously matters a lot to those of us responsible for creating one. The better the early design, the less work needed refining it later. That is what we are really talking about, is it not.

What I am arguing for is not less software design, but a realistic software design process. We need to recognize the difference between *designing software* and *a software design*. It makes sense to use any tools and techniques that we find help us during the design process. It does not make sense to forget what is the real software design. When we have worked out some aspect of software design, we should not let incorrect comparisons with hardware engineering disciplines keep us from correctly documenting our work in a software design. *YES, WE SHOULD CODE IT*. If what we are really doing is software design, then everything we do will somehow be reflected in code. We might as well write the code (or that portion of it that makes sense) when we make the decisions that affect that code.

I know all the arguments for "language independent" software design notations. They all ignore a fundamental problem. Software design involves translating concepts from some problem space into a programming language. This translation has to be done by human beings, and since our programming languages are usually totally inadequate to express the concepts of the problem space directly, it is usually a difficult and error prone process. When we translate concepts from one form to another, especially complex ones, we often loose important information. If several translations are involved, we are likely to end up with a final product that lost too much vital information, that does not accurately reflect our original concept, and/or that simply contains errors. This is compounded several times when the people actually doing the translation are different for each step. Remember, there is nothing sacred about C++ (or Ada, or C, or Smalltalk, or LISP, or any programming language). It is not the native language of our computers. Fundamentally, programming languages are just a design notation themselves. I do not see any point in introducing extra translation steps if they can be avoided.

There are a couple of problems with my "code as design" approach, which I acknowledge. The first is that even the best programming languages have serious weaknesses as tools for expressing certain aspects of a software design. The information is in the code (if it isn't, then it wasn't software design information) but it is very difficult to get it out in human readable form. These are the "other aspects" of a software design mentioned above. The second problem is similar. There is going to be information from the problem space that went into the software design, but that can not be reconstructed from the software design itself. We want to capture this information in case we need to change the software design later. The typical source code comment is not an adequate mechanism.

Both of these problems mean that auxiliary documentation is as important to software design as it is to any other engineering discipline, if not more so. We must recognize auxiliary documentation as such, however, and not confuse it with the software design.

What we really need is more expressive programming languages. This is what led to my statement about C++ being a major advance in software design art. C++ is a more expressive programming language, which makes it a better software design tool.

As a final topic, consider what my point of view reveals about traditional software development processes. Ultimately, all software design processes end up validating and refining the final design via a build/test cycle. Any pretense otherwise is just silliness. Yet, traditional MIL-STD and other waterfall model development processes will not even allow writing one line of code until a certain tonnage of auxiliary documentation has been produced and reviewed. Often, the people who produce this documentation then go on to other things leaving a group of new, and usually much younger and less experienced people to actually generate the real software design. It is hardly surprising (to me anyway) that this process has fallen into such disrepute that no real developers advocate it. What are they trying instead?

Now we have rapid prototyping and spiral development. In my view, it is easy to see why these are replacing the waterfall method. Both of these are just excuses for writing code earlier in the development cycle so that the process of refining the design via build/test can begin sooner. They also typically get the same people involved in both the top level design and the actual code design. Not surprisingly, these two approaches are both seen as significant improvements. Even the best of the traditional approaches continue to try to break software design into disjoint steps with separate notations and products, and then they continue to wonder why they have problems getting a final software product that is correct.

Projects done in Ada have shown significant improvements in the time necessary to integrate, test and debug (at the expense of some extra top level design effort). Structured programming was considered a breakthrough in its time. Object oriented design and C++ are taking the world by storm. Forget all the explanations offered for these phenomenon, and consider what hasn't worked. CASE tools? Popular, yes; universal, no. Structure charts? Same thing. Likewise, Warner-Orr diagrams, Booch diagrams, object diagrams, you name it. Each has its strengths, and a single fundamental weakness - it really isn't a software design. Ultimately, improvements in programming techniques are overwhelmingly more important to software development than anything else.

There seems to be a collective fantasy of the software community that if we could just find the right graphical design notation so that software designs would look like other engineering designs, then we could take our place as software engineers alongside the other disciplines. I disagree. Engineering is about how you do the process, not about whether the final product needs a CAD system to render it. We in the software business are so close, but so far away. Software is so -- well, *soft*. A software system can represent

anything. This, plus the economics of the build/test cycle, makes it very unlikely that we will find general purpose methods for validating a software design other than the current trial and error. We can improve the process, however. Maybe if we started treating software development as a homogeneous design process, and concentrated on improving the most important phases (programming, debug and test), we might find our industry to be more of a disciplined science than we think it is.

I still do not know if I have made my point, but in summary:

- Real software is what runs on computers. This means that real software is not C++ (or any other programming language).
- Real software is built by computers (via compilers and linkers).
- This means that a source code listing (in any programming language) is really a software design.
- It follows from the above that real software is incredibly cheap to build, and getting cheaper all the time as computers get faster.
- Software is incredibly expensive to design. This is true in both an absolute sense (because of its ever increasing complexity), and relative to the cost of a software build.
- Programming is a design activity - a good software design process recognizes this and doesn't hesitate to code when coding makes sense.
- Coding actually makes sense a whole lot more often than traditional software design processes would have you believe.
- Testing and debugging are design activities - they are the software equivalent of the analysis, simulation, modeling, and testing phases of other engineering disciplines. The goal is to validate and improve the design before the final product is built. A good software design process recognizes this and doesn't try to disown or short change the steps.
- There are other design activities - call them top level design, module design, or whatever. A good software design process recognizes this and deliberately includes the steps.
- All design activities interact. A good software design process recognizes this and allows the design to change, sometimes radically, as other design steps reveal the need.

- Many different software design notations are potentially useful - as auxiliary documentation (it would be nice to have some tools that help us generate and maintain auxiliary documentation automatically from the actual source code. This would be particularly useful in maintaining a graphical representation of the structural aspects of the design).
- Ultimately, real advances in software development depend upon advances in programming. C++ is a good step in the right direction. For software engineering's sake, we need a great many more steps.

###

## Historical Note

In an email correspondence discussing the publication of this letter and its relationship to the original letter,

> "The sequence of events was as follows: I read an article about software design in one issue of C++ Journal. I sent a letter to the editor complaining about something (I don't remember what, and can not find the letter on my system these days). The editor printed the letter, and a response. What I have attached is the letter I emailed to the editor as a response to his response. I think you will find it rather familiar. I personally find it better written than the article itself. The editor Livleen Singh offered to allow me to turn my points into a full article, which I did."

## About the Author

Jack W. Reeves is a senior software developer with over 30 years experience in the industry. He has worked on systems ranging from simulators for the space shuttle, military command and control systems, air traffic control systems, medical imaging systems, financial data distribution systems, embedded systems, drivers, and utilities. He has exclusively been an OO developer for the last 15 years.